Second-Order Classical Sequent Calculus

Philip Johnson-Freyd

Abstract

We present a sequent calculus that allows to abstract over a type variable. The introduction of the abstraction on the right-hand side of a sequent corresponds to the *universal* quantification; whereas, the introduction of the abstraction on the left-hand side of a sequent corresponds to the *existential* quantification. The calculus provides a correspondence with second-order classical propositional logic. To show consistency of the logic and consequently of the type system, we prove the property of strong normalization. The proof is based on an extension of the reducibility method, consisting of associating to each type two sets: a set of of strong normalizing terms and a set of strong normalizing co-terms.

1 Introduction

A persistent challenge in programming language design is combining features. Many features have been proposed that are of interest individually, but the interplay of features can be non-obvious. Particularly troublesome can be efforts to intermix dynamic features such as powerful control operators with static features such as advanced type systems: How do we design a language which combines Scheme's continuations with ML's modules? How do C#'s generators interact with its generic types? Should the difference in evaluation order between ML and Haskell lead to differences in their type systems?

In order to reason about programming languages, it is often helpful to design core calculi that exhibit the features we are interested in. By rigorously analyzing these calculi we can gain important insights of importance not just to language design and implementation but also to reasoning about programs. A particularly important source of inspiration for the design of programming language calculi is the deep connection between programming languages and proof systems for formal logics. The switch from intuitionistic logic to classical logic gives insight into what happens when continuations are added to a language. While natural deduction serves to model many high level languages, sequent calculus helps in theorizing lower level aspects of computation and gives insight into questions such as the duality between evaluation strategies and the relationship between objects and algebraic data types. Programming language features related to abstraction and polymorphism can be studied by analogy to second order quantifiers in logic. While Java's generics or ML's

polymorphism can be understood in terms of universal quantifiers, implementation hiding in module systems is related to the use of existential quantifiers.

We study the combination of control effects and type abstraction by describing a calculus that is in a "proofs as programs correspondence" with second-order classical propositional logic. Section 2, provides a brief background in proof theory and its connection to computation, culminating in the presentation of a calculus which provides an elegant computational interpretation for first-order classical propositional logic. In Section 3, we present a variant of System F [7] in sequent calculus style; polymorphism comes with a computational interpretation. In Section 4, we review the role of existential types to provide modularity, and show how they can be easily incorporated into our sequent framework by following a duality argument. In Section 5, we motivate strong normalization as a technique for establishing the consistency of second-order classical propositional logic, and then we give a proof of the strong normalization property using a variant of Girard's "Reducibility Candidates" method. We conclude in Section 6.

2 Proofs as Programs

We review propositional logic in sequent style by introducing the reader to the use of *left* and *right* rules, as opposed to the use of *introduction* and *elimination* rules of natural deduction. The right rules allow one to derive new conclusions, whereas the left rules allow one to derive new assumptions. We then present a calculus or language which encodes such sequent proofs.

2.1 Sequent calculus

In the years of 1934 and 1935 Gerhard Gentzen published his "Untersuchungen ber das logische Schlieen"[6]. Gentzen described a system of *natural deduction* for both classical and intuitionistic predicate logics. Natural deduction was as "close as possible to actual reasoning" [6]. Further, Gentzen posited his 'Hauptsatz'[1] that every proof in natural deduction could be reduced to a normal form. Although not published (but see [13]), Gentzen claimed a direct proof of this fact for intuitionistic natural deduction, but he could not determine a way to reduce *classical* proofs that made use of the law of the excluded middle. For this he introduced a new system called *sequent calculus* and showed the equivalence of classical and intuitionistic sequent calculi to natural deduction as well as to a "Hilbert style" system. For sequent calculus, even in the classical case, Gentzen was able to show his Hauptsatz in the form of "cut elimination." It is hard to overstate the significance of this result: the 'Hauptsatz' is the principle theorem in proof theory. It implies, among other things, that these logics are *internally consistent* in that they are unable to prove the proposition *false*.

[1] "Main Theorem."

We will start with first-order propositional logic, in which a formula is built from propositional variables p, q, r, *etc*, and the implication connective:

$$P, Q \in \text{Formulae} ::= P \to Q \mid p \ .$$

For example, $p \to q$ and $p \to (q \to r)$ are formulae. Whereas, a formula in which we use some unspecified formula, like $P \to P$, is more properly called a *formula scheme*. However, in the following we refer to a formula scheme as a formula. The fundamental judgment of sequent calculus is written

$$\Gamma \vdash \Delta$$

where Γ (the assumptions) and Δ (the conclusions) are sets of formulae. The intuitive interpretation of the above sequent is that "if every formula in Γ is true then at least one formula in Δ is true." The simplest rule of sequent calculus is the *axiom*, which says that any formula entails itself:

$$\frac{}{\Gamma, P \vdash P, \Delta}$$

The ability to work with lemmas is fundamental to mathematics. In sequent calculus, lemmas are supported by way of the cut rule:

$$\frac{\Gamma \vdash P, \Delta \qquad \Gamma, P \vdash \Delta}{\Gamma \vdash \Delta}$$

To define logical connectives in the sequent calculus we must do two things: describe how to prove a sequent which has that connective in its conclusion, and, describe how to prove a sequent which has that connective as an assumption.

We now consider the *implication* connective. We should be able to prove $P \to Q$ if we can prove Q under the assumption P. The *right introduction rule* for implication encodes this view:

$$\frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \to Q, \Delta}$$

On the other hand, the *left introduction rule* for implication must show us how to prove a sequent of the form $\Gamma, P \to Q \vdash \Delta$. We know that something in Δ holds under the assumption of $P \to Q$ if something in Δ holds under the assumption Q or if either P or Δ holds:

$$\frac{\Gamma, Q \vdash \Delta \qquad \Gamma \vdash P, \Delta}{\Gamma, P \to Q \vdash \Delta}$$

We have given a short semantic justification for the left and right rules for implication, but one might still wonder how can we be sure that the inference rules are correct. In other

words, how can we guarantee that the logic does not allow one to derive a false proposition. Gentzen proposed *cut elimination* as the primary correctness criteria. Suppose we have a proof that ends in a right introduction of $P \to Q$ and another proof that ends in a left introduction of the same formula. A cut between these two proofs can be eliminated in so far as it can be reduced to a proof that only makes use of cuts at structurally smaller formula:

$$\dfrac{\dfrac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \to Q, \Delta} \qquad \dfrac{\Gamma, Q \vdash \Delta \qquad \Gamma \vdash P, \Delta}{\Gamma, P \to Q \vdash \Delta}}{\Gamma \vdash \Delta}$$

$$\Downarrow$$

$$\dfrac{\Gamma \vdash P, \Delta \qquad \dfrac{\Gamma, P \vdash Q, \Delta \qquad \Gamma, Q \vdash \Delta}{\Gamma, P \vdash \Delta}}{\Gamma \vdash \Delta}$$

The cut elimination procedure works to eliminate all cuts from our proofs. The existence of cut elimination means that if something is provable, it is provable without the use of cuts. This property ensures the consistency of the logic: what can be shown about a connective is exactly what can be shown by the left and right rules only. So for example, we can convince ourselves that the formula $A \to B$ is not provable by only considering the right rule for implication:

$$\dfrac{A \vdash B}{\vdash A \to B}$$

Since one cannot derive B from A we are done. Note however that if we would not be able to remove all cuts from a proof, then we would need to consider also the case below:

$$\dfrac{\vdash C \qquad C \vdash A \to B}{\vdash A \to B}$$

Observe that cut elimination is inherently non deterministic. For example, we could give an alternative cut elimination for implication:

$$\dfrac{\dfrac{\Gamma \vdash P, \Delta \qquad \Gamma, P \vdash Q, \Delta}{\Gamma \vdash Q, \Delta} \qquad \Gamma, Q \vdash \Delta}{\Gamma \vdash \Delta}$$

In addition to the left and right rules for connectives, there are the *structural rules*. We should not have to care about the order of assumptions and conclusions in our sequent,

$$P, Q \in \text{Formulae} ::= P \to Q \mid p$$

$$\frac{\Gamma \vdash P, \Delta \qquad \Gamma, P \vdash \Delta}{\Gamma \vdash \Delta} \qquad \Gamma, P \vdash P, \Delta$$

$$\frac{\Gamma, P \vdash \Delta \qquad \Gamma \vdash Q, \Delta}{\Gamma, P \to Q \vdash \Delta} \qquad \frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \to Q, \Delta}$$

Figure 1: First-order classical propositional logic in sequent style

and thus Gentzen gives the rules of exchange:

$$\frac{\Gamma, P, Q, \Gamma' \vdash \Delta}{\Gamma, Q, P, \Gamma' \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, P, Q, \Delta'}{\Gamma \vdash \Delta, Q, P, \Delta'}$$

The weakening rules state that if a sequent is provable, the same sequent is provable with any number of extra assumptions and conclusions:

$$\frac{\Gamma \vdash \Delta}{\Gamma, \Gamma' \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \Delta'}$$

While the rules of contraction state that two assumptions (conclusions) of the same formula can be shared:

$$\frac{\Gamma, P, P \vdash \Delta}{\Gamma, P \vdash \Delta} \qquad \frac{\Gamma \vdash P, P, \Delta}{\Gamma \vdash P, \Delta}$$

Uses of the structural rules are often kept implicit. In particular, because we treat Γ and Δ as *sets* rather than purely formal sequences of formulae, exchange and contraction are completely automatic. We summarize the sequent calculus rules in Figure 1.

Unlike the single conclusion natural deduction, which is by nature *intuitionistic*, sequent calculus with multiple conclusions corresponds to *classical logic*. This can be seen in the derivation of Peirce's law–$((P \to Q) \to P) \to P$–a statement, which when added as an axiom, has the consequence of turning intuitionistic logic into classical logic.

$$\dfrac{\dfrac{\dfrac{}{P \vdash P} \qquad \dfrac{\dfrac{}{P \vdash P, Q}}{\vdash P \to Q, P}}{(P \to Q) \to P \vdash P}}{\vdash ((P \to Q) \to P) \to P}$$

2.2 Term assignment for first-order classical propositional logic

We are now looking at the rules of Figure 1 not as inference rules but as typing rules, instead of talking about the implication $P \to Q$ we will talk about a function type. We introduce the notion of a *command*, say c, which corresponds to the sequent:

$$c : \Gamma \vdash \Delta \ ,$$

where Γ and Δ give the types of the free variables appearing in c. Note, however, that we now have two kinds of variables: the normal variables which stand for an unknown term, and the co-variables which stand for continuations or co-terms. Terms and co-terms correspond to a sequent having a distinguished formula on the right-hand and left-hand side of the sequent, respectively. They are typed with the following judgements:

$$\Gamma \vdash v : T \mid \Delta \qquad \Gamma \mid e : T \vdash \Delta$$

A term v (co-term e) has type T provided that its free variables occur in Γ and its free co-variables occur in Δ. The construction of a command $\langle v \| e \rangle$ is interpreted as a cut. The right introduction rule for implication is captured by a lambda abstraction. Whereas, the left introduction rule for implication is captured by a stack extension operator $\cdot$, reminiscent of Lisp cons constructor. Given a command c we also need a way to focus on either a conclusion or an assumption. This is the role of the constructs $\mu\alpha.c$ and $\tilde{\mu}x.c$. The term $\mu\alpha.c$ is similar to Felleisen $\mathcal{C}$ control operator, it allows one to name the context [5] or to specify which output channel one wants to observe the answer of a command. The dual construct $\tilde{\mu}x.c$ abstracts over a variable in a command to construct a co-term and can be thought of as an open let expression (*i.e.,* . of the form $\mathsf{let}\ x = \Box\ \mathsf{in}\ t$), it specifies which channel one wants to observe the input of a command. The full syntax for the Curien-Herbelin calculus is in Figure 2 while the static semantics is in Figure 3. The reduction rules are given in Figure 4. To get an intuition for these rules, we consider a simple translation from the lambda calculus.

$$\begin{aligned} [\![\lambda x.v]\!] &= \lambda x.[\![v]\!] \\ [\![x]\!] &= x \\ [\![v_1\ v_2]\!] &= \mu\alpha.\, \langle [\![v_1]\!] \| [\![v_2]\!] \cdot \alpha \rangle \end{aligned}$$

Under this translation, we can see that the one step lambda reduction

$$(\lambda x.v_1)\ v_2 \longrightarrow v_1[v_2/x]$$

$$
\begin{aligned}
T, S \in \text{Types} &::= T \to S \mid t \\
e \in \text{Co-Terms} &::= \alpha \mid v \cdot e \mid \tilde{\mu} x.c \\
v \in \text{Terms} &::= x \mid \lambda x.e \mid \mu\alpha.c \\
c \in \text{Commands} &::= \langle v \| e \rangle
\end{aligned}
$$

Figure 2: Syntax of $\mu\tilde{\mu}^{\to}$

$$
\frac{\Gamma \vdash v : T \mid \Delta \qquad \Gamma \mid e : T \vdash \Delta}{\langle v \| e \rangle : \Gamma \vdash \Delta}[\text{cut}]
$$

$$
\frac{}{\Gamma, x : T \vdash x : T \mid \Delta}[\text{Var}] \qquad \frac{}{\Gamma \mid \alpha : T \vdash \alpha : T, \Delta}[\text{Co-Var}]
$$

$$
\frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x : A.v : A \to B}[\to \text{R}] \qquad \frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid v \cdot e : A \to B \vdash \Delta}[\to \text{L}]
$$

$$
\frac{c : (\Gamma \vdash \alpha : T, \Delta)}{\Gamma \vdash \mu\alpha.c : T \mid \Delta}[\mu] \qquad \frac{c : (\Gamma, x : T \vdash \Delta)}{\Gamma \mid \tilde{\mu} x.c : T \vdash \Delta}[\tilde{\mu}]
$$

Figure 3: Typing rules of $\mu\tilde{\mu}^{\to}$

becomes a multi step reduction.

$$
\begin{aligned}
[\![(\lambda x.v_1)\ v_2]\!] &= \mu\alpha.\, \langle [\![\lambda x.v_1]\!] \| [\![v_2]\!] \cdot \alpha \rangle \\
&= \mu\alpha.\, \langle \lambda x.[\![v_1]\!] \| [\![v_2]\!] \cdot \alpha \rangle \\
\langle \mu\alpha.\, \langle \lambda x.[\![v_1]\!] \| [\![v_2]\!] \cdot \alpha \rangle \| e \rangle &\longrightarrow \langle \lambda x.[\![v_1]\!] \| [\![v_2]\!] \cdot e \rangle \\
&\longrightarrow \langle [\![v_2]\!] \| \tilde{\mu} x.\, \langle [\![v_1]\!] \| e \rangle \rangle \\
&\longrightarrow \langle [\![v_1]\!][[\![v_2]\!]/x] \| e \rangle \\
&= \langle [\![v_1[v_2/x]]\!] \| e \rangle
\end{aligned}
$$

It is remarkable that the sequent calculus provides also a typing for the Krivine abstract machine, which is a simple model for the call-by-name evaluation of lambda calculus [2].

Notice that input and output abstraction produce a critical pair which leads to losing confluence:

$$
(\langle x \| \delta \rangle \longleftarrow_{\mu} \langle \mu\alpha.\, \langle x \| \delta \rangle \| \tilde{\mu} x.\, \langle y \| \delta \rangle \rangle \longrightarrow_{\tilde{\mu}} \langle y \| \delta \rangle
$$

If we interpret the reduction rules as defining an equational theory, it is possible to prove

$$\langle \mu\alpha.c \| e \rangle \longrightarrow c[e/\alpha] \qquad (\mu)$$
$$\langle v \| \tilde{\mu}x.c \rangle \longrightarrow c[v/x] \qquad (\tilde{\mu})$$
$$\langle \lambda x.v_1 \| v_2 \cdot e \rangle \longrightarrow \langle v_2 \| \tilde{\mu}x.\, \langle v_1 \| e \rangle \rangle \qquad (\beta)$$

Figure 4: Dynamic Semantics of $\mu\tilde{\mu}^{\rightarrow}$

that $\langle x \| \delta \rangle$ and $\langle y \| \delta \rangle$ are equal. In other words, the consequence of the critical pair is the collapse of the equational theory: every term is equal to every other term.

Intermezzo 1. To restore confluence, Curien and Herbelin [2] restrict the syntax into two subsystems: one corresponding to a call-by-name language and the other to a call-by-value language. An alternative design, taken by Downen and Ariola [3], is to modify the reduction rules for input and output abstraction to only fire in the presence of *syntactic values*, that is, we restrict the μ and $\tilde{\mu}$ rules as follows:

$$\langle \mu\alpha.c \| E \rangle \longrightarrow c[E/\alpha]$$
$$\langle V \| \tilde{\mu}x.c \rangle \longrightarrow c[V/x]$$

A call-by-name reduction theory can then be formed by making the class of values as large as possible while excluding $\tilde{\mu}$ from the class of co-values:

$$V, V_1, V_2, \ldots \in \text{Values} ::= \lambda x.v \mid x \mid \mu\alpha.c$$
$$E, E_1, E_2, \ldots \in \text{Co-Values} ::= v \cdot E \mid \alpha$$

Dually, a call-by-value reduction theory can be formed by making the class of co-values as large as possible while excluding μ from the class of values.

$$V, V_1, V_2, \ldots \in \text{Values} ::= \lambda x.v \mid x$$
$$E, E_1, E_2, \ldots \in \text{Co-Values} ::= v \cdot e \mid \alpha \mid \tilde{\mu}x.c$$

Making the notion of values a parameter of the theory allows Downen and Ariola to also give a call-by-need reduction theory that uses the same notion of value as call-by-value but which further restricts the notion of co-value. In here however we will focus on the unrestricted calculus, which we call $\mu\tilde{\mu}^{\rightarrow}$.

3 Universal Quantification

As we did before, we will first look at the logic point of view, and then switch to the computational interpretation of the universal quantification.

3.1 Proof Theory of $\forall$

The significant change needed to our language is that the syntax for formulae now allows quantification over propositional variables:

$$P, Q \in \text{Formulae} ::= P \to Q \mid t \mid \forall(t.P) \ .$$

In first-order propositional logic, one can say $p \to p$, which could stand for the sentence "if today is Friday then today is Friday", now however we can also say $\forall p.p \to p$ which means "for every proposition p, proposition p implies itself." Hence, we have *second-order* instead of *first-order*. Second-order propositional logic is more restricted than first-order predicate logic because all predicate symbols have arity zero (since it does not expect any arguments). On other other hand, second-order propositional logic is more general than first-order predicate logic because it allows quantifications over propositions. With respect to the inference rules: the left rule for the universal quantifier formalizes the idea that if we can prove our goal under the assumption that some formula in some specific case holds, then we can prove the same goal under the (usually stronger) assumption that the formula is true in all cases.

$$\frac{\Gamma, P[Q/t] \vdash \Delta}{\Gamma, \forall(t.P) \vdash \Delta}$$

The challenge is in giving the right rule. What we would like to be able to say is that if we can prove some formula P which includes some proposition variables, then we can prove that formula holds for any valuation of those variables.

$$\frac{\Gamma \vdash P, \Delta}{\Gamma \vdash \forall(t.P), \Delta}[\text{unsafe}]$$

The problem is that we have to be careful about scoping, that is unfortunately always the case once bound variables are introduced. The above naive version of the right rule does not ensure that the proposition variable t being used is fresh, and so allows for proving things which are false. This can be seen easily if we extend the logic with the connective $\bot$ (pronounced "false") which has no right introduction rule but can be always introduced on the left, and $\top$ (pronounced "true") which has no left introduction rule but can be always introduced on the right.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{t \vdash t}}{t \vdash \forall(t.t)}}{\vdash t \to \forall(t.t)}}{\vdash \forall(t.t \to \forall(t.t)))} \qquad \dfrac{\dfrac{}{\vdash \top, \bot} \qquad \dfrac{\dfrac{}{\bot \vdash \bot}}{\forall(t.t) \vdash \bot}}{\top \to \forall(t.t) \vdash \bot}}{\forall(t.t \to \forall(t.t)) \vdash \bot}$$

$$\vdash \bot$$

The key to avoiding this problem is to ensure that all uses of the proposition variables are properly scoped. To to this, we modify the inference rules to track which proposition variables are in scope. The judgments are extended by the addition of a context Θ which is a set of proposition variables and is written adjacent and below the turnstile. We consider a judgment syntactically well formed only if all free proposition variables occur in Θ, for example, the following judgment:

$$t \vdash t$$

is not valid. We define the notation Θ, t to be the extension of Θ with the new proposition variable t under the assumption that t is not in Θ.

$$\Theta, t \stackrel{\Delta}{=} \Theta \cup \{t\} \qquad \text{when } t \notin \Theta$$

$$\Theta, t \stackrel{\Delta}{=} \text{undefined} \qquad \text{when } t \in \Theta$$

The use of Θ allows us to give the rules of the universal connective in a safe way:

$$\frac{\Gamma, P[Q/t] \vdash_\Theta \Delta}{\Gamma, \forall(t.P) \vdash_\Theta \Delta} \qquad \frac{\Gamma \vdash_{\Theta,t} P, \Delta}{\Gamma \vdash_\Theta \forall(t.P), \Delta}$$

It is easy to check that the addition of the Θ and the resulting modified rules avoid the problematic derivation from earlier.

$$\dfrac{\dfrac{\dfrac{???}{t \vdash_t \forall(t.t)}}{\vdash_t t \to \forall(t.t)}}{\vdash \forall(t.t \to \forall(t.t)))} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \top, \bot}}{\top \to \forall(t.t) \vdash \bot} \qquad \dfrac{\dfrac{}{\bot \vdash \bot}}{\forall(t.t) \vdash \bot}}{\forall(t.t \to \forall(t.t)) \vdash \bot}$$

$$\vdash \bot$$

There is no proof of the sequent $t \vdash_t \forall(t.t)$ because the right introduction rule for the universal is only defined if the proposition variable being introduced is fresh. Note however that since we only consider formulae up to α equivalence, we could rename $\forall(t.t)$ to $\forall(s.s)$. We are now able to apply the right rule as shown below:

$$\frac{t \vdash_{t,s} s}{t \vdash_t \forall(s.s)}$$

However, we are now stuck since there is no proof of the sequent $t \vdash_{t,s} s$. We give the second-order propositional logic in sequent style in Figure 5.

Remark 1. To emphasize the difference between first-order and second-order propositional logic, consider the formula $P \to P$. In first-order propositional logic, one can say that

$$P, Q \in \text{Formulae} ::= P \to Q \mid p \mid \forall(p.P)$$

$$\frac{\Gamma \vdash_\Theta P, \Delta \qquad \Gamma, P \vdash_\Theta \Delta}{\Gamma \vdash_\Theta \Delta} \qquad \frac{}{\Gamma, P \vdash_\Theta P, \Delta}$$

$$\frac{\Gamma, Q \vdash_\Theta \Delta \qquad \Gamma \vdash_\Theta P, \Delta}{\Gamma, P \to Q \vdash_\Theta \Delta} \qquad \frac{\Gamma, P \vdash_\Theta P, \Delta}{\Gamma \vdash_\Theta P \to Q, \Delta}$$

$$\frac{\Gamma, P[Q/t] \vdash_\Theta \Delta}{\Gamma, \forall(t.P) \vdash_\Theta \Delta} \qquad \frac{\Gamma \vdash_{\Theta,t} P, \Delta}{\Gamma \vdash_\Theta \forall(t.P), \Delta}$$

Figure 5: Second-order classical propositional logic in sequent style

$P \to P$ is a tautology for every formula P. Notice that the quantification is at the meta-level, whereas in second-order case quantification is made formal: the formula $\forall p.p \to p$ is a tautology:

$$\dfrac{\dfrac{p \vdash p}{\vdash p \to p}}{\vdash \forall p.p \to p}$$

3.2 Term assignment for $\forall$

It turns out that the computational interpretation of the universal quantification is *polymorphism*, that is the capability of writing *uniform* programs that work with a different number, even infinite, of types. In the $\mu\tilde{\mu}^{\to}$ calculus it is possible to write an identity function at the type $t \to t$, where t stands for an unknown type. However, we cannot then use the identity at different types. We enrich the set of types with polymorphic types:

$$T, S \in \text{Types} \ ::= T \to R \mid t \mid \forall(t.T)$$

We can thus define an identity function for all types, which we call id:

$$\vdash \Lambda t.\lambda x.x : \forall(t.t \to t)$$

We extend the judgments of $\mu\tilde{\mu}^{\to}$ to incorporate a set of type variables. In each case we consider the judgment syntactically well formed only if all free type variables occur in Θ.

$$\Gamma \vdash_\Theta v : T \mid \Delta \qquad \Gamma \mid e : T \vdash_\Theta \Delta \qquad c : (\Gamma \vdash_\Theta \Delta)$$

We then have to give a term assignment for the left and right rules for $\forall$. The term for the right rule mirrors the term assignment for the arrow type, the only difference being

$$T, S \in \text{Types} \ ::= T \to R \mid t \mid \forall(t.T)$$

$$\begin{aligned} e \in \text{Co-Terms} &::= \alpha \mid \tilde{\mu}x.c \mid v.e \mid T.e \\ v \in \text{Terms} &::= x \mid \mu\alpha.c \mid \lambda x.v \mid \Lambda t.v \end{aligned}$$

$$\frac{\Gamma \mid e : T[S/t] \vdash_\Theta \Delta}{\Gamma \mid S.e : \forall(t.T) \vdash_\Theta \Delta}[\forall_L] \qquad \frac{\Gamma \vdash_{\Theta,t} v : T \mid \Delta}{\Gamma \vdash_\Theta \Lambda t.v : \forall(t.T) \mid \Delta}[\forall_R]$$

$$\langle \Lambda t.v \| T.e \rangle \longrightarrow \langle v[T/t] \| e \rangle$$

Figure 6: Syntax, static and dynamic semantics of $\mu\tilde{\mu}^{\to,\forall}$

that now we are waiting for a type instead of a term. The co-term for the left rule also is similar to the left rule for the arrow type, the only difference being that we are waiting for a type and the type of the remaining context depends on this type. The syntax, typing rules and the dynamic semantics are given in Figure 6.

One interesting example of the power of $\mu\tilde{\mu}^{\to,\forall}$ is to apply the identity function to itself producing a new term of type $\forall(t.t \to t)$. There are two ways to do this. The first, and more direct, approach is made possible by the fact that $\mu\tilde{\mu}^{\to,\forall}$ is *impredicative*. That is, type abstraction abstracts over *all* types, not just simple types. In particular, we can instantiate the polymorphic identity function with the type of the polymorphic identity function.

$$\mu\alpha.\, \langle \mathtt{id} \| \forall(t.t \to t).\mathtt{id} \cdot \alpha) \rangle$$

The second way is to instantiate the identity function with two different types: once with the desired type, and once with the function type.

$$\Lambda t.\mu\alpha.\, \langle \mathtt{id} \| t \to t.(\mu\beta.\, \langle \mathtt{id} \| t.\beta \rangle \cdot \alpha) \rangle]$$

In programming languages, such as Haskell or ML, polymorphism is implicit and so there is no way to differentiate between the two ways of applying the identity function to itself. The first implementation we gave works by specializing and then generalizing types corresponds to the Hindley-Milner type elaboration these languages use [10]. Although Haskell will permit a polymorphic type be assigned to the program that applies the identity function to itself, ML will not.

System F in sequent calculus style is also considered by Summers and van Bakel, who describe a classical sequent calculus with implication and universal connectives [14]. Their

calculus, called $\mathcal{X}$, is quite different from $\mu\tilde{\mu}^{\rightarrow,\forall}$. Unlike $\mu\tilde{\mu}^{\rightarrow,\forall}$, $\mathcal{X}$ lacks any notion of term or co-term, making everything into commands. Further, $\mathcal{X}$ does not use substitutions in its reduction theory. Lengrand et al. present a sequent calculus approach to *Pure Type Systems* [8]. They go significantly beyond the present work in considering not only polymorphism, but the entire family of Pure Type Systems including dependent types. Their PTS system is intuitionistic (*i.e.,* it does not incorporate control operators) while $\mu\tilde{\mu}^{\rightarrow,\forall}$ is classical. Further their interpretation of cuts as explicit substitutions differs significantly from the interpretation of cuts in the $\mu\tilde{\mu}^{\rightarrow,\forall}$. In particular, because it is intuitionistic, cuts are interpreted as explicit substitutions of terms in for variables, and it does not expose the dualities of $\mu\tilde{\mu}^{\rightarrow,\forall}$. Lengrand and Miquel develop a sequent calculus presentation of a non-confluent classical variant of System F_ω and show it to be strongly normalizing [9]. They utilize a one sided presentation of the sequent calculus instead of our two sided presentation.

4 Existential Quantification

As we did for universal quantification, we will first look at the proof theory side, and then consider the programming language feature that provides a computational reading of the the existential quantification.

4.1 Proof theory for $\exists$

We consider one more formula of the form $\exists p.P$, as for example we can write $\exists p.p \rightarrow p$. To better illustrate the advantage of using a sequent style framework, we will first consider the inference rules for existential in natural deduction style. Whereas for universal quantification, the introduction rule was hard to define and the elimination rule was easy, for the existential quantification the role is flipped: the introduction rule for existential is easy but the elimination rule is hard to define. The existential introduction is expressed as :

$$\frac{\Gamma \vdash P[Q/p]}{\Gamma \vdash \exists p.P}$$

In the sequent style, the rule is very similar, we simply add the context for the co-variables:

$$\frac{\Gamma \vdash P[Q/p], \Delta}{\Gamma \vdash \exists(p.P), \Delta}$$

Consider now the elimination rule:

$$\frac{\Gamma \vdash \exists p.P \qquad \Gamma, P \vdash Q}{\Gamma \vdash Q}$$

which has two proviso: p cannot occur free in Γ and in Q. To understand the role of these proviso let us consider some examples.

Example 1. If one relaxes the proviso that the quantified variable can occur in the conclusion, then we could have the following wrong derivation:

$$\dfrac{\dfrac{\exists p.p \vdash \exists p.p \qquad \exists p.p, p \vdash p}{\exists p.p \vdash p}\ \exists_e}{\exists p.p \vdash \forall p.p}\ \forall_i$$

Example 2. Let us first introduce the proof by contradiction rule (where contradiction is represented by the constant $\bot$) and the negation elimination rule, respectively:

$$\dfrac{\Gamma, \neg P \vdash \bot}{\Gamma \vdash P}\ RAA \qquad \dfrac{\Gamma \vdash \neg \quad \Gamma \vdash P}{\Gamma \vdash \bot}\ \neg_e$$

With these rules now consider this derivation:

$$\dfrac{\dfrac{\exists p.p, \neg p \vdash \exists p.p \qquad \dfrac{\exists p.p, \neg p, p \vdash \neg p \qquad \exists p.p, \neg p, p \vdash p}{\exists p.p, \neg p, p \vdash \bot}\ \neg_e}{\exists p.p, \neg p \vdash \bot}\ \exists_e}{\dfrac{\exists p.p \vdash p}{\exists p.p \vdash \forall p.p}\ \forall_i}\ RAA$$

The wrong step consists in making the assumption p when p already occurs in the assumptions.

We now represent the elimination rule in sequent style. To avoid the problem of variable clashing we keep track of the propositional variables in the set θ, as we did for the universal quantification. The left rule becomes:

$$\dfrac{\Gamma, P \vdash_{\Theta, p} \Delta}{\Gamma, \exists(p.P) \vdash_{\Theta} \Delta}$$

The notation θ, p captures the fact that p cannot occur in Γ and Δ. We present the full system with both quantifiers in Figure 7. Notice the advantage of using a sequent style framework: the rules for the existential quantification come out automatically as dual to the universal quantification.

Example 3. Let us represents the wrong derivations we did before in sequent style. Consider the first example, we have:

$$\dfrac{\dfrac{p \vdash_p p}{\exists p.p \vdash_p p}\ \exists_L}{\exists p.p \vdash \forall p.p}\ \forall_R$$

The existential to the left is not correct since p already occurs free in the conclusion.

$$\frac{\Gamma, P[Q/p] \vdash_\Theta \Delta}{\Gamma, \forall(p.P) \vdash_\Theta \Delta} \qquad \frac{\Gamma \vdash_{\Theta,t} P, \Delta}{\Gamma \vdash_\Theta \forall(p.P), \Delta}$$

$$\frac{\Gamma \vdash_\Theta P[Q/p], \Delta}{\Gamma \vdash_\Theta \exists(p.P), \Delta} \qquad \frac{\Gamma, P \vdash_{\Theta,t} \Delta}{\Gamma, \exists(p.P) \vdash_\Theta \Delta}$$

Figure 7: First-order propositional quantification

Example 4.

$$\dfrac{\dfrac{\exists p.p \vdash_p \exists p.p \quad \dfrac{p \vdash_p p}{\exists p.p \vdash_p p}\ \exists_l}{\exists p.p \vdash_p p}\ cut}{\exists p.p \vdash \forall p.p}\ \forall_i$$

As before, the existential on the left is not valid since p occurs in the Θ.

4.2 Term assignment for $\exists$

It turns out that the computational interpretation of the existential quantification corresponds to the notion of an abstract data type. A single piece of functionality might be implemented in multiple ways. For example, an abstraction representing sets of integers might be implemented as a linked list, or, alternatively as red-black tree. A client program using only the abstract type of a set of integers can be written independently of the concrete implementation. To be concrete, let us consider a simple example. Take a module `COLORS` [4] that exposes two constants `red` and `black` and a function `show` for turning these colors into strings. In an ML or Haskell like language, one implementation of this module `COLORS` might use booleans to store the color.

$$\begin{aligned}
&\texttt{booleanColor} = \texttt{COLORS}\ \{\\
&\qquad \texttt{red} = \texttt{true}\\
&\qquad \texttt{blue} = \texttt{false}\\
&\qquad \texttt{show} = \lambda x.\mathbf{if}\ x\ \mathbf{then}\text{“}red\text{”}\mathbf{else}\text{“}blue\text{”}\\
&\qquad \}
\end{aligned}$$

An alternative representation might encode the colors not as booleans, but as the strings.

$$\begin{aligned}\texttt{stringColor} = \texttt{COLORS}\ \{ \\ \texttt{red} = \text{“}red\text{”} \\ \texttt{blue} = \text{“}blue\text{”} \\ \texttt{show} = \lambda x.x \\ \}\end{aligned}$$

Intuitively, these two implementations are in some sense equivalent. That is, if we have a program that uses one, we would like to be able to substitute it in the other and know that nothing about the observable behavior of the program changes. The problem is that of course, we can not simply perform such a substitution. There may well be programs that depend on *implementation details* of the `booleanColor` module. The program

$$\textbf{if}\ \texttt{booleanColor.red}\ \textbf{then}\ 1\ \textbf{else}\ 2$$

produces the output 1 while the program

$$\textbf{if}\ \texttt{stringColor.red}\ \textbf{then}\ 1\ \textbf{else}\ 2$$

has a type error. The key to enabling modularity and avoiding this problem is existential types [11]. The *signature* of a module is an existential type and the implementation is a term of that type. Because types constrain the set of possible contexts that a module may be used, it can be used to prevent programs from determining implementation details. In our example, we could assign a type which would make the two implementations observationally indistinguishable.

$$\begin{aligned}\texttt{COLORS} = \exists t.\{ \\ \texttt{red} : t \\ \texttt{blue} : t \\ \texttt{show} : t \to \texttt{String} \\ \}\end{aligned}$$

Such a type would outlaw the program which distinguishes between the modules since the projection `booleanColor.red` does not yield a term of boolean type, but rather a term whose type is given as a type variable. In this way, the client of the module is generally unable to determine what implementation the module is using. This implementation hiding is important for practical programming, since it allows for software to be developed in pieces independently of each other, and, for components to be replaced. Because the type system enforces abstraction, these development practices are safe: we can prove that no clients of a module depend on some implementation detail of that module by simply examining the type of the module.

$$
\begin{array}{lll}
T, S \in \text{Types} & ::= & t | T \to S | \forall(t.T) | \exists(t.T) \\
e \ldots \in \text{Co-Terms} & ::= & \alpha | \tilde{\mu}x.c | v.e | T.e | \Lambda t.e \\
v \ldots \in \text{Terms} & ::= & x | \mu\alpha.c | \lambda x.v | T.v | \Lambda t.v
\end{array}
$$

$$
\frac{\Gamma \vdash v : T \mid \Delta \qquad \Gamma \mid e : T \vdash \Delta}{\langle v \| e \rangle : \Gamma \vdash \Delta}[\text{cut}]
$$

$$
\frac{}{\Gamma, x : T \vdash x : T \mid \Delta}[\text{Var}] \qquad \frac{}{\Gamma \mid \alpha : T \vdash \alpha : T, \Delta}[\text{Co-Var}]
$$

$$
\frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x : A.v : A \to B}[\to \text{R}] \qquad \frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid v \cdot e : A \to B \vdash \Delta}[\to \text{L}]
$$

$$
\frac{c : (\Gamma \vdash \alpha : T, \Delta)}{\Gamma \vdash \mu\alpha.c : T \mid \Delta}[\mu] \qquad \frac{c : (\Gamma, x : T \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : T \vdash \Delta}[\tilde{\mu}]
$$

$$
\frac{\Gamma \mid e : T[S/t] \vdash_\Theta \Delta}{\Gamma \mid S.e : \forall(t.T) \vdash_\Theta \Delta}[\forall_L] \qquad \frac{\Gamma \vdash_{\Theta, t} v : T \mid \Delta}{\Gamma \vdash_\Theta \Lambda t.v : \forall(t.T) \mid \Delta}[\forall_R]
$$

$$
\frac{\Gamma \vdash_\Theta v : T[S/t] \mid \Delta}{\Gamma \vdash_\Theta S.v : \exists(t.T) \mid \Delta}[\exists_R] \qquad \frac{\Gamma \mid e : T \vdash_{\Theta, t} \Delta)}{\Gamma \mid \Lambda t.e : \exists(t.T) \vdash_\Theta \Delta}[\exists_L]
$$

$$
\begin{array}{lll}
\langle \mu\alpha.c \| e \rangle & \longrightarrow c[e/\alpha] & (\mu) \\
\langle v \| \tilde{\mu}x.c \rangle & \longrightarrow c[v/x] & (\tilde{\mu}) \\
\langle \lambda x.v_1 \| v_2 \cdot e \rangle & \longrightarrow \langle v_2 \| \tilde{\mu}x.\, \langle v_1 \| e \rangle \rangle & (\beta) \\
\langle \Lambda t.v \| T.e \rangle & \longrightarrow \langle v[T/t] \| e \rangle & (\forall) \\
\langle S.v \| \Lambda t.e \rangle & \longrightarrow \langle v \| e[T/t] \rangle & (\exists)
\end{array}
$$

Figure 8: Syntax, static and dynamic semantics of $\mu\tilde{\mu}^{\to,\forall,\exists}$

We now enrich our set of types:

$$
T, S \in \text{Types} ::= t | T \to S | \forall(t.T) | \exists(t.T)
$$

and add the following typing rules:

$$
\frac{\Gamma \vdash_\Theta v : T[S/t] \mid \Delta}{\Gamma \vdash_\Theta S.v : \exists(t.T) \mid \Delta}[\exists_R] \qquad \frac{\Gamma \mid e : T \vdash_{\Theta, t} \Delta}{\Gamma \mid \Lambda t.e : \exists(t.T) \vdash_\Theta \Delta}[\exists_L]
$$

The existential on the left corresponds to an abstraction over the propositional variable, whereas the existential on the right corresponds to a pair of a type (the "witness") and the body of the module, that is, the implementation of the module. Notice the duality with the universal quantification. Instead in natural deduction, as Pierce comments on [12], the introduction and elimination forms for existential types are syntactically heavier

than the simple type abstraction and application associated to universals. We present the full system which we write $\mu\tilde{\mu}^{\rightarrow,\forall,\exists}$,in Figure 8.

Remark 2. To enhance the reader intuition, we present the term assignment for the existential quantification in natural deduction and show its translation in sequent style. We have:

$$\frac{\Gamma \vdash_\theta v : T[S/t]}{\Gamma \vdash_\theta \mathsf{pack}(S, v) : \exists t.T} \qquad \frac{\Gamma \vdash_\theta v_1 : \exists t.T \qquad \Gamma, x : T \vdash_{\theta,t} v_2 : S}{\Gamma \vdash_\theta \mathsf{open}\ v_1\ \mathsf{as}\ (t, x)\ \mathsf{in}\ v_2 : S}$$

The dynamic semantics for call-by-value is:

$$\mathsf{open}\ \mathsf{pack}(S, V)\ \mathsf{as}\ (t, x)\ \mathsf{in}\ v \quad \rightarrow \quad v[S/t, V/x]$$

The translation in sequent calculus is:

$$\begin{array}{lcl} [\![\mathsf{pack}(S, v)]\!] & = & S.[\![v]\!] \\ [\![\mathsf{open}\ v_1\ \mathsf{as}\ (t, x)\ \mathsf{in}\ v_2]\!] & = & \mu\alpha.\, \langle [\![v_1]\!] \| \Lambda t.\tilde{\mu}x.\, \langle [\![v_2]\!] \| \alpha \rangle \rangle \end{array}$$

Notice how the translation preserves the semantics:

$$\mu\alpha.\, \langle S.[\![V]\!] \| \Lambda t.\tilde{\mu}x.\, \langle [\![v]\!] \| \alpha \rangle \rangle \ \rightarrow\ \mu\alpha.\, \langle [\![V]\!] \| \tilde{\mu}x.\, \langle [\![v]\!] \| \alpha \rangle\, [S/t] \rangle \ \rightarrow\ \mu\alpha.\, \langle [\![v]\!][S/t][[\![V]\!]/x] \| \alpha \rangle$$

We present in sequent calculus some examples taken from [12]. Consider the package p_4 be

$$\mathsf{pack}(\mathsf{nat}, (0, \lambda x : \mathsf{nat}.x + 1) : \exists t.t * (t \rightarrow \mathsf{nat})\ ,$$

the term

$$\mathsf{open}\ p_4\ \mathsf{as}\ (t, x)\ \mathsf{in}\ \mathsf{snd}\ (\mathsf{fst}\ x)$$

evaluates to 1. This is expressed in the sequent as an interaction between the producer of the module and the consumer or client of the module:

$$\langle \mathsf{nat}.(0, \lambda x : \mathsf{nat}.x + 1) \| \Lambda t.\tilde{\mu}x.\, \langle x \| \mathsf{snd}.\tilde{\mu}f.\, \langle f \| \mu\alpha.\, \langle x \| \mathsf{fst}.\alpha \rangle\, .\delta \rangle \rangle \rangle$$

Note that if we changed the type of p_4 to $\exists t.t * (t \rightarrow t)$ then the above command would not type since type t will escape its scope. The same happens with the command

$$\langle \mathsf{nat}.(0, \lambda x : \mathsf{nat}.x + 1) \| \Lambda t.\mathsf{snd}.\tilde{\mu}x.\, \langle x + 1 \| \alpha \rangle \rangle\ ,$$

which will raise a typing error since the type of x is t (the client of the package does not know the witness type) and therefore $x + 1$ does not type checks. Another error is raised with the program:

$$\langle \mathsf{nat}.(0, \lambda x : \mathsf{nat}.x + 1) \| \Lambda t.\mathsf{fst}.\alpha \rangle$$

The context has this typing judgement:

$$\Lambda t.\mathsf{fst}.\alpha : \exists t.t * (t \rightarrow t) \vdash_t \alpha : t$$

Since t occurs in Δ, one would have:

$$\mathsf{fst}.\alpha : s * (s \to s) \vdash_{t,s} \alpha : t$$

which is not derivable. If the above term were typable then we would have:

$$\mu\alpha.\, \langle \mathsf{nat}.0 \| \Lambda s.s \rangle$$

which will lead to having an inhabitant of the type $\forall t.t$: $\Lambda s.\mu\alpha.\, \langle \mathsf{nat}.0 \| \Lambda s.s \rangle$.

5 Strong Normalization

We would like to prove that $\mu\tilde{\mu}^{\to,\forall,\exists}$ is strongly normalizing: if a command is typeable it is not the start of any infinite reduction sequences. From the perspective of computation, strong normalization ensures that all programs terminate. From the perspective of logic, strong normalization allows us to prove internal consistency. Internal consistency is the proposition that there exists an unprovable formula.

Theorem 1. *Second order propositional logic is internally consistent.*

Proof. Consider the sequent $\vdash_t t$. It is enough to show that there is no proof of this sequent. Assume, by contradiction, that there was such a proof. Then, there would have to be some command $c_1 : (\vdash_t \alpha : t)$. Any command c_1 reduces to also has the type $\vdash_t \alpha : t$ (type safety). In particular, by strong normalization any reduction sequence starting at c_1 must terminate, meaning there must be some $c_2 : (\vdash_t \alpha : t)$ such that c_2 is not the start of any reduction sequence. For $\langle v \| e \rangle$ to be typeable and not the start of any reduction sequence, one of v or e must be a (co)-variable. Since $\langle v \| e \rangle\, (\vdash_t \alpha : t)$, and $\vdash_t x : T \mid \alpha : t$ is not derivable, we know that e is a co-variable. Indeed, the only possibility is that $e = \alpha$. Thus, we know $\langle v \| \alpha \rangle : (\vdash_t \alpha : t)$ meaning $\vdash_t v : t \mid \alpha : t$. Since t is a type variable, the only available rule for v is the output abstraction rule $\mu\beta.c'$, but if $\langle \mu\alpha.c' \| \alpha \rangle \longrightarrow c'[\alpha/\beta]$ which violates our assumption that $\langle v \| e \rangle$ is not the start of any reduction sequence. As such, there is no command of type $(\vdash_t \alpha : t)$ and so no proof of $\vdash_t t$. □

One way we might go about proving strong normalization, would be to come up with some "measure" in the form of a natural number that we assign to every command, and show that during reduction this number is strictly decreasing. Unfortunately, such a strategy is unlikely to work. Strong normalization for $\mu\tilde{\mu}^{\to,\forall}$ would imply strong normalization for System F which in turn would imply the consistency of second-order Peano arithmetic [18], and each of those steps can be shown by simple induction. Thus, because a combinatorial proof of strong normalization for $\mu\tilde{\mu}_S^{\to,\forall}$ would presumably be formalizable using the axioms of arithmetic, it could be used to construct a proof of the consistency of Peano arithmetic from within Peano arithmetic. This, of course, is outlawed by Gödel's incompleteness theorem (and the consistency of arithmetic). Instead, we adopt a proof technique

that is more *semantic* in character based on Girard's "reducibility candidates" method [7]. As such, we freely take advantage of the non-arithmetic reasoning available in set theory.

The idea is to provide a semantic interpretation of the meaning of types. By carefully constructing the semantics we can ensure two things: that the typing rules are sound with respect to the semantics, and that semantically well typed programs are strongly normalizing.

In order to simplify the presentation, we focus on the steps necessary to get the proof to work for implication. Our proof extends to handle $\forall$ and $\exists$ using standard techniques [7]. The proof for $\mu\tilde{\mu}^{\rightarrow}$ allows us to capture those aspects of the proof that are unique to the setting of $\mu\tilde{\mu}$.

There are many possible approaches to providing a denotational semantics to a programming language–here we use the free one. We will simply interpret $\mu\tilde{\mu}$ in terms of itself: terms will be interpreted as terms, co-terms as co-terms, and commands as commands. The reader may be surprised that such an approach to semantics would be useful, indeed, it does not seem obvious that it deserves to be called "semantics" at all. For our purposes though, these "term models" are interesting. That is because what we are interested in is the semantic interpretation of *types* in order to prove a property (namely strong normalization) stated in terms of the *operational* semantics of terms.

In languages such as the lambda calculus where there is only one syntactic category of terms, term models work by interpreting types as *sets of terms*. In the sequent calculus though, types don't just interpret terms but also co-terms. Thus, the semantic function $[\![-]\!]$ interprets types as pairs of sets of terms and sets of co-terms.

$$[\![T]\!] \in \mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term})$$

Given the interpretation of types, we can define what it means for a term to be *semantically* well typed. In place of the syntactic judgment

$$\Gamma \vdash v : T \mid \Delta$$

we have the semantic judgment

$$\Gamma \vDash v : T \mid \Delta$$

where Γ and Δ are sets of (co-)variables together with types. Variables stand-in for possible substitutions, so the meaning of $\Gamma \vDash v : T \mid \Delta$ must be that for any substitution ϕ that implements Γ and Δ, applying ϕ to v (which we will, in a slight abuse of notation, write $\phi(v)$) must be in the set of terms which is associated with T.

$$\Gamma \vDash v : T \mid \Delta \quad \stackrel{\Delta}{=} \quad \forall \psi \in [\![\Gamma]\!], \psi \in [\![\Delta]\!] \Rightarrow \psi(v) \in \mathbf{fst}([\![\mathrm{T}]\!])$$

Here $\psi \in [\![\Gamma]\!]$ means that ψ maps variables in Γ to terms in the sets associated with those variable types.

$$[\![\Gamma]\!] = \{\psi : \text{Var} \rightarrow \text{Term} | \forall (x : T) \in \Gamma, \psi(x) \in \mathbf{fst}([\![\mathrm{T}]\!])\}$$
$$[\![\Delta]\!] = \{\psi : \text{Co-Var} \rightarrow \text{Co-Term} | \forall (\alpha : T) \in \Gamma, \psi(\alpha) \in \mathbf{snd}([\![\mathrm{T}]\!])\}$$

Similarly, we can give a semantic typing judgment for co-terms and commands. We use the notation SN for the set of strongly normalizing commands.

$$\begin{aligned} \Gamma \mid e : T \vDash \Delta &\triangleq \forall \psi \in [\![\Gamma]\!], \psi \in [\![\Delta]\!] \Rightarrow \psi(e) \in \mathbf{snd}([\![\mathrm{T}]\!]) \\ c : (\Gamma \vDash \Delta) &\triangleq \forall \psi \in [\![\Gamma]\!], \psi \in [\![\Delta]\!] \Rightarrow \psi(c) \in SN \end{aligned}$$

We wish to establish that *soundness*, that is, if $\Gamma \vdash v : T \mid \Delta$ is *derivable*, then the semantic judgment $\Gamma \vDash v : T \mid \Delta$ is *true* (and the equivalent statements for co-terms and commands).

The basic way to show soundness will be to induct over all the possible proofs. In order to make induction work there are certain conditions that must hold on the output of our semantic function $[\![-]\!]$. We call a pair that satisfies these conditions a *reducibility candidate* and the set of reducibility candidates CR. For example, one rule that we will have to consider is the cut rule.

$$\frac{\Gamma \vdash v : T \mid \Delta \qquad \Gamma \mid e : T \vdash \Delta}{\langle v \| e \rangle : \Gamma \vdash \Delta}$$

Semantically, the cut rule tells us that:

$$(\Gamma \vDash v : T \mid \Delta) \wedge (\Gamma \mid e : T \vDash \Delta) \Rightarrow (c : (\Gamma \vDash \Delta))$$

Expanding this definition out yields:

$$\begin{aligned} &(\forall \psi \in [\![\Gamma]\!], \psi \in [\![\Delta]\!] \Rightarrow \psi(v) \in \mathbf{fst}([\![\mathrm{T}]\!])) \\ \wedge &(\forall \psi \in [\![\Gamma]\!], \psi \in [\![\Delta]\!] \Rightarrow \psi(e) \in \mathbf{snd}([\![\mathrm{T}]\!])) \\ &\Rightarrow (\forall \psi \in [\![\Gamma]\!], \psi \in [\![\Delta]\!] \Rightarrow \psi(c) \in SN) \end{aligned}$$

Our first condition on reducibility candidates then is simply the condition needed to make this always hold.

CR 1. *(Orthogonality) For any term v and co-term e such that $v \in \boldsymbol{fst}([\![T]\!])$ and $e \in \boldsymbol{snd}([\![T]\!])$ the command $\langle v \| e \rangle$ is strongly normalizing.*

Another useful property would be the semantic equivalent of subject reduction. If a (co-)term is semantically well typed, then anything it internally reduces to should be semantically well typed also.

CR 2. *Forward Closure:*

1. *If $v \in \boldsymbol{fst}([\![T]\!])$ and $v \longrightarrow v'$ then $v' \in \boldsymbol{fst}([\![T]\!])$.*
2. *If $e \in \boldsymbol{snd}([\![T]\!])$ and $e \longrightarrow e'$ then $e' \in \boldsymbol{snd}([\![T]\!])$.*

In addition to the conditions we need to ensure soundness, we want the model to ensure that all typed commands are strongly normalizing. But $c : (\Gamma \vDash \Delta)$ only ensures that for any $\psi \in [\![\Gamma]\!] \wedge \psi \in [\![\Delta]\!]$ the substitution command is strongly normalizing: $\psi(c) \in SN$. To make sure c itself is strongly normalizing we need to make sure that the identity substitution is in $[\![\Gamma]\!]$ and $[\![\Delta]\!]$.

CR 3. *Inclusion of (Co-)Variables:*

1. $Var \subseteq \boldsymbol{fst}([\![T]\!])$ *and*
2. $Co\text{-}Var \subseteq \boldsymbol{snd}([\![T]\!])$.

Finally, returning to the conditions needed for soundness, the set of terms and the set of co-terms should include some uses of μ and $\tilde{\mu}$, respectively. The abstraction rule:

$$\frac{c : (\Gamma \vdash \alpha : T, \Delta)}{\Gamma \vdash \mu\alpha.c : T \mid \Delta}$$

has the semantic interpretation that

$$c : (\Gamma \vDash \alpha : T, \Delta) \Rightarrow \Gamma \vDash \mu\alpha.c : T \mid \Delta$$

which expands to:

$$(\forall \psi \in [\![\Gamma]\!], \psi \in [\![\alpha : T, \Delta]\!] \Rightarrow \psi(c) \in SN) \Rightarrow (\forall \psi \in [\![\Gamma]\!], \psi \in [\![\Delta]\!] \Rightarrow \mu\alpha.\psi(c) \in \mathbf{fst}([\![T]\!])$$

This gives our last condition on reducibility candidates.

CR 4. *Saturation:*

1. $\{\mu\alpha.c | \forall e \in \boldsymbol{snd}([\![T]\!]), c[e/\alpha] \in SN\} \subseteq \boldsymbol{fst}([\![T]\!])$;
2. $\{\tilde{\mu}x.c | \forall v \in \boldsymbol{fst}([\![T]\!]), c[v/x] \in SN\} \subseteq \boldsymbol{snd}([\![T]\!])$.

To summarize, types are to be interpreted as reducibility candidates and a reducibility candidate is a pair of a set of terms and a set of co-terms that satisfies the conditions of

- **CR 1**: orthogonality,
- **CR 2**: forward closure,
- **CR 3**: inclusion of (co-)variables and
- **CR 4**: saturation.

Lemma 1. *If (A, B) is a reducibility candidate then all terms in A and co-terms in B are internally strongly normalizing.*

Proof. Let $v \in A$. We need to show that v is strongly normalizing. (A, B) is a reducibility candidate, thus it satisfies **CR 3** (inclusion of (co-)variables) so $\alpha \in B$. Further, since (A, B) is a reducibility candidate, it must satisfy **CR 1** (orthogonally) meaning the command $\langle v \| \alpha \rangle$ is strongly normalizing. By contradiction, assume v was not strongly normalizing, then there would be an infinite reduction sequence $v \longrightarrow v_1 \longrightarrow v_2 \longrightarrow \ldots$ but that would imply the existence of an infinite reduction sequence $\langle v \| \alpha \rangle \longrightarrow \langle v_1 \| \alpha \rangle \longrightarrow \langle v_2 \| \alpha \rangle \longrightarrow \ldots$ which is a contradiction. Dually, if $e \in B$ then we know $x \in A$ and that the command $\langle x \| e \rangle$ is SN so e must be internally strongly normalizing. □

Given a command $\langle v \| e \rangle$ we can divide the possible reductions into those that would happen *internally* to v and e and those that happen at the *head* and so involve both v and e. A command is strongly normalizing precisely if all the commands it reduces to are strongly normalizing. The next lemma allows us to consider only head reduction in certain cases

Lemma 2. *Let A be a set of terms and B a set of co-terms, then (A, B) is orthogonal $(\forall v \in A, \forall e \in B, \langle v \| e \rangle \in SN)$ if:*

1. *A and B are forward closed,*
2. *all terms in A and co-terms in B are strongly normalizing, and*
3. *$\forall v \in A, \forall e \in B$ for any command c such that $\langle v \| e \rangle$ head reduces to c we have that $c \in SN$.*

Proof. Given $v, v' \in A$ and $e, e' \in B$ we say $(v, e) > (v', e')$ if either

1. $v \longrightarrow^\star v'$ and $e \longrightarrow^+ e'$ or
2. $e \longrightarrow^\star e'$ and $v \longrightarrow^+ v'$.

It is easy to see that $>$ is well founded on $A \times B$ precisely because both A and B are internally strongly normalizing. Now, we wish to show that for all $v \in A$ and $e \in B$ the command $\langle v \| e \rangle \in SN$. We do this by Noetherian induction on (v, e) using $>$ as the well order. $\langle v \| e \rangle$ is strongly normalizing if every thing it reduces to is strongly normalizing. There are three possible reductions:

1. $\langle v \| e \rangle \longrightarrow \langle v' \| e \rangle$ where $v \longrightarrow v'$. Since A is forward closed $v' \in A$ and since $(v, e) > (v', e)$, $\langle v' \| e \rangle$ is strongly normalizing by the inductive hypothesis.
2. $\langle v \| e \rangle \longrightarrow \langle v \| e' \rangle$ where $e \longrightarrow e'$. Since B is forward closed $e' \in B$ and since $(v, e) > (v, e')$, $\langle v \| e' \rangle$ is strongly normalizing by the inductive hypothesis.
3. $\langle v \| e \rangle \longrightarrow c$ by head reduction, so $c \in SN$ by assumption.

Note that our use of Noetherian induction avoids the need to consider the base case explicitly. □

Now that we know what must be true about the interpretation of every type, we can consider what must be true about interpretation of specific types. We will assume that the function $[\![-]\!]$ is defined at all type variables, and yields a reducibility candidate (that is, a pair of sets satisfying the four conditions). We require no additional conditions for the candidate defined for type variables because there are no introduction or elimination rules for type variables. By contrast, we will *define* the meaning of $[\![T_1 \to T_2]\!]$. In particular, in order to ensure soundness, we must support the left introduction rule for implication.

$$\{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\} \subseteq \mathbf{snd}([\![T_1 \to T_2]\!])$$

Similarly, we need to incorporate the right rule for implication.

$$\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!]))\} \subseteq \mathbf{fst}([\![T_1 \to T_2]\!]))$$

Based on what we have seen so far, we can give a pair of mutually recursive equations for $[\![T_1 \to T_2]\!]$

$$\begin{aligned}
\mathbf{fst}([\![T_1 \to T_2]\!])) &= \text{Var} \\
&\quad \cup \{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!])), v[v'/x] \in \mathbf{fst}([\![T_2]\!]))\} \\
&\quad \cup \{\mu\alpha.c | \forall e \in \mathbf{snd}([\![T_1 \to T_2]\!]), c[e/\alpha] \in SN\} \\
\mathbf{snd}([\![T_1 \to T_2]\!]) &= \text{Co-Var} \\
&\quad \cup \{v \cdot e | v \in \mathbf{fst}([\![T_1]\!])), e \in \mathbf{snd}([\![T_2]\!])\} \\
&\quad \cup \{\tilde{\mu}x.c | \forall v \in \mathbf{fst}([\![T_1 \to T_2]\!])), c[v/x] \in SN\}
\end{aligned}$$

This is not enough to define $[\![T_1 \to T_2]\!]$ since we don't know how many solutions these equations have (if they have any at all). To get around this, we define a function $F_{T_1 \to T_2}$

$$\begin{aligned}
F_{T_1 \to T_2} &: \mathcal{P}(\text{Term}) \times \mathcal{P}(\textbf{Co-Term}) \to \mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term}) \\
F_{T_1 \to T_2}(A, B) &= (\text{Var} \\
&\quad \cup \{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\} \\
&\quad \cup \{\mu\alpha.c | \forall e \in B, c[e/\alpha] \in SN\} \\
&\quad , \text{Co-Var} \\
&\quad \cup \{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\} \\
&\quad \cup \{\tilde{\mu}x.c | \forall v \in A, c[v/x] \in SN\})
\end{aligned}$$

Indeed, we could generalize this construction. Let S be an arbitrary pair of a set of

terms and a set of co-terms.

$$
\begin{aligned}
F_S &: \mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term}) \to \mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term}) \\
F_S(A,B) &= (\text{Var} \\
&\quad \cup \mathbf{fst}(S) \\
&\quad \cup \{\mu\alpha.c | \forall e \in B, c[e/\alpha] \in SN\} \\
&\quad , \text{Co-Var} \\
&\quad \cup \mathbf{snd}(S) \\
&\quad \cup \{\tilde{\mu}x.c | \forall v \in A, c[v/x] \in SN\})
\end{aligned}
$$

We recover the definition of $F_{T_1 \to T_2}$ by setting $S = (\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\}, \{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\})$.

Examining F_S we see that $\mathbf{fst}(F_{T_1 \to T_2}(A,B))$ is determined entirely by B. What is more, $\mathbf{fst}(F_S(A,-)$ is monotonically decreasing (order reversing) in terms of the ordinary ordering of sets as subset inclusion. Similarly, the second component of $F_S(A,B)$ is determined entirely by A and is monotonically decreasing in A. We can thus define the ordering relation $\sqsubseteq$ on the elements of $\mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term})$ as the product of the usual set theoretic inclusion in the first component and the dual of the usual set theoretic ordering in the second.

$$(A,B) \sqsubseteq (A',B') \triangleq A \subseteq A' \wedge B' \subseteq B$$

Lemma 3. *F_S is monotonic with respect to $\sqsubseteq$.*

Proof. Suppose $(A,B) \sqsubseteq (A',B')$. We wish to show that $F_S(A,B) \sqsubseteq F_S(A',B')$. By the definition of $\sqsubseteq$ we know $A \subseteq A'$ and that $B' \subseteq B$. Given any command c such that $\forall e \in B, c[e/\alpha] \in SN$ it must be the case that $\forall e \in B', c[e\alpha] \in SN$. Therefore, the set of terms $\{\mu\alpha.c | \forall e \in B, c[e/\alpha] \in SN\}$ is a subset of $\{\mu\alpha.c | \forall e \in B', c[e/\alpha] \in SN\}$. Dually, $\{\tilde{\mu}x.c | \forall v \in A', c[v/x] \in SN\} \subseteq \{\tilde{\mu}x.c | \forall v \in A, c[v/x] \in SN\}$. Thus:

$$
\begin{aligned}
\mathbf{fst}(F_S(A,B)) &= \text{Var} \cup \mathbf{fst}(S) \\
&\quad \cup \{\mu\alpha.c | \forall e \in B, c[e/\alpha] \in SN\} \\
&\subseteq \text{Var} \cup \mathbf{fst}(S) \\
&\quad \cup \{\mu\alpha.c | \forall e \in B', c[e/\alpha] \in SN\} \\
&\subseteq \mathbf{fst}(F_S(A',B')) \\
\mathbf{snd}(F_S(A',B')) &= \text{Co-Var} \cup \mathbf{snd}(S) \\
&\quad \cup \{\tilde{\mu}x.c | \forall v \in A', c[v/x] \in SN\} \\
&\subseteq \text{Co-Var} \cup \mathbf{snd}(S) \\
&\quad \cup \{\tilde{\mu}x.c | \forall v \in A, c[v/x] \in SN\} \\
&\subseteq \mathbf{snd}(F_S(A,B))
\end{aligned}
$$

$\mathbf{fst}(F_S(A,B)) \subseteq \mathbf{fst}(F_S(A',B'))$ and $\mathbf{snd}(F_S(A',B')) \subseteq \mathbf{snd}(F_S(A,B))$. Therefore $F_S(A,B) \sqsubseteq F_S(A',B')$. □

Lemma 4. *F_S has a fixed point, and indeed a least fixed point, when ordered by $\sqsubseteq$.*

Proof. Since $\mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term})$ ordered by $\sqsubseteq$ is the product partial orders of two partial orders each of which is a complete lattice, $\mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term})$ is a complete lattice ordered by $\sqsubseteq$. Therefore, by the Knaster-Tarski theorem [17] F_S has a least fixed point. □

We can now give the semantic interpretation of types. To do this, we will use an auxiliary semantic function $(\!|-|\!)$ which we will use to *seed* the fixed point construction.

$$\begin{aligned}
(\!|-|\!) &: \text{Type} \to \mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term}) \\
(\!|t|\!) &= (\emptyset, \emptyset) \\
(\!|T_1 \to T_2|\!) &= (\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\} \\
&\quad , \{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\})
\end{aligned}$$

From the seed we can define the interpretation of types ($[\![-]\!]$) by way of the fixed point. For our proof to work it does not matter which fixed point we take for $[\![\text{T}]\!]$, but the least fixed point is available and defined.

$$[\![T]\!] = \mathbf{lfp}_{\sqsubseteq} F_{(\!|T|\!)}$$

Since $[\![T]\!]$ is a fixed point of $F_{(\!|T|\!)}$ we know

$$\begin{aligned}
\mathbf{fst}([\![T]\!]) &= \text{Var} \cup \mathbf{fst}((\!|T|\!)) \\
&\quad \cup \{\mu\alpha.c | \forall e \in \mathbf{snd}([\![T]\!]), c[e/\alpha] \in SN\} \\
\mathbf{snd}([\![T]\!]) &= \text{Co-Var} \cup \mathbf{snd}((\!|T|\!)) \\
&\quad \cup \{\tilde{\mu}x.c | \forall v \in \mathbf{fst}([\![T]\!]), c[v/x] \in SN\}
\end{aligned}$$

Remark 3. We showed in Lemma 4 that $F_{T_1 \to T_2}$ had a least fixed point, but it may build intuition to work through the construction of this fixed point. In particular, observe that given any set of $Q \in \mathcal{P}(\mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term})$ we have:

$$\mathbf{sup}\{F_{T_1 \to T_2}(T) | T \in Q\} = F_{T_1 \to T_2}(\mathbf{sup}\ Q)$$

Which is to say that $F_{T_1 \to T_2}$ preserves limits. Thus, the least fixed point guaranteed to exist by the Knaster-Tarski theorem is also given constructively by Kleene's fixed point theorem as the limit of an increasing chain.

The least element of the domain $\mathcal{P}(\text{Term}) \times \mathcal{P}(\text{Co-Term})$ ordered by $\sqsubseteq$ is the pair

$$\bot = (\emptyset, \textbf{Co-Term})$$

We can then define the upward chain of candidates $C_0, C_1, \ldots$.

$$\begin{aligned} C_0 &= \bot \\ C_{n+1} &= F_{T_1 \to T_2}(C_n) \end{aligned}$$

The least upper bound is then given as the supremum of this chain.

$$\begin{aligned} [\![T_1 \to T_2]\!] &= \textbf{lfp}_{\sqsubseteq}(F_{T_1 \to T_2}) \\ &= \textbf{sup}\{C_n | n \in \mathbb{N}\} \\ &= (\bigcup_n \textbf{fst}(C_n), \bigcap_n \textbf{snd}(C_n)) \end{aligned}$$

Lemma 5. *Whenever $(\!|T|\!)$ is internally strongly normalizing so is $[\![T]\!]$*

Proof. Assume $(\!|T|\!)$ is internally strongly normalizing. If $v \in \textbf{fst}([\![T]\!])$ then either

- $v \in$ Var in which case it is internally strongly normalizing as there are no possible reductions,
- $v \in \textbf{fst}((\!|T|\!))$ and so internally strongly normalizing by assumption, or
- $v \in \{\mu\alpha.c | \forall e \in \textbf{snd}([\![T]\!]), c[e/\alpha] \in SN\}$ and since $\alpha \in \textbf{snd}([\![T]\!])$ (by definition) we know that $v = \mu\alpha.c$ where $c = c[\alpha/\alpha[$ is strongly normalizing.

If $e \in \textbf{snd}([\![T]\!])$ then either

- e is a variable and so strongly normalizing,
- $e \in \textbf{fst}((\!|T|\!))$ and so internally strongly normalizing by assumption, or
- $e \in \{\tilde{\mu}x.c | \forall v \in \textbf{fst}([\![T]\!]), c[v/x] \in SN\}$ where since $x \in \textbf{fst}([\![T]\!])$ we know e is strongly normalizing.

□

Lemma 6. *If $(\!|T|\!)$ satisfies the forward closure property then so does $[\![T]\!]$*

Proof. If $v \in \textbf{fst}([\![T]\!])$ and $v \longrightarrow v'$ then $v' \in \textbf{fst}([\![T]\!])$.

- If v is a variable than there does not exists a v' such that $v \longrightarrow v'$.
- If $v \in \textbf{fst}((\!|T|\!))$ and $v \longrightarrow v'$ then by assumption $v' \in \textbf{fst}((\!|T|\!))$ and so $v' \in [\![T]\!]$

- If $v = \mu\alpha.c$ then for any $e \in \mathbf{snd}(\llbracket T \rrbracket)$ we know $c[e/\alpha] \in SN$. Suppose $v \longrightarrow v'$ then $v' = \mu\alpha.c'$ where $c \longrightarrow c'$ meaning that for any $e \in \mathbf{snd}(\llbracket T \rrbracket)$, $c[e/\alpha] \longrightarrow c'[e/\alpha]$ and so $c'[e/\alpha] \in SN$. Thus $\mu\alpha.c' \in \mathbf{fst}(\llbracket T \rrbracket)$.

Dually, if $e \in \mathbf{snd}(\llbracket T \rrbracket)$ and $e \longrightarrow e'$ then $e' \in \mathbf{snd}(\llbracket T \rrbracket)$.

- If e is a co-variable then there is no possible e' such that $e \longrightarrow e'$.
- If $e \in \mathbf{snd}(\llparenthesis T \rrparenthesis)$ and $e \longrightarrow e'$ then by assumption $e' \in \mathbf{snd}(\llparenthesis T \rrparenthesis)$ and so $e' \in \llbracket T \rrbracket$.
- If $e = \tilde{\mu}x.c$ then $\forall v \in \mathbf{fst}(\llbracket T \rrbracket), c[v/x] \in SN$ so if $c \longrightarrow c'$ then $\forall v \in \mathbf{fst}(\llbracket T \rrbracket), c'[v/x] \in SN$ so $\tilde{\mu}x.c' \in \mathbf{snd}(\llbracket T \rrbracket)$.

□

We define a *nucleus* as a pair (A, B) where A is a set of terms and B is a set of co-terms such that

1. A and B are internally strongly normalizing;
2. A and B are forward closed;
3. there are no terms of the form $\mu\alpha.c$ in A and no co-terms of the form $\tilde{\mu}x.c$ in B;
4. (A, B) is orthogonal, that is $\forall v \in A, \forall e \in B, \langle v \| e \rangle \in SN$.

The notion of nucleus allows us to abstract out what we need to prove about $\llparenthesis T \rrparenthesis$ in order to show that $\llbracket T \rrbracket$ is a reducibility candidate.

Lemma 7. *If $\llparenthesis T \rrparenthesis$ is a nucleus then $\llbracket T \rrbracket$ is orthogonal.*

Proof. We wish to show that $v \in \mathbf{fst}(\llbracket T \rrbracket)$ and $e \in \mathbf{snd}(\llbracket T \rrbracket)$ implies that $\langle v \| e \rangle \in SN$ while we know that $\llparenthesis T \rrparenthesis$ is a nucleus. We know by Lemma 6 that $\mathbf{fst}(\llbracket T \rrbracket)$ and $\mathbf{snd}(\llbracket T \rrbracket)$ are forward closed under reduction, and by Lemma 5 that these internal reduction relations are well founded. Thus, by Lemma 2 We need only consider head reduction. There are only three forms of head reduction to consider.

1. $\langle \mu\alpha.c \| e \rangle \longrightarrow c[e/\alpha]$. Here, $c[e/\alpha]$ is strongly normalizing since we know that $\mathbf{fst}(\llparenthesis T \rrparenthesis)$ does not contain any term of the form $\mu\alpha.c$ (because it is a nucleus) if $v = \mu\alpha.c$ we know by the definition of $\mathbf{fst}(\llbracket T \rrbracket)$ that $\forall e \in \mathbf{snd}(\llbracket T \rrbracket), c[e/\alpha] \in SN$.
2. $\langle v \| \tilde{\mu}x.c \rangle \longrightarrow c[v/x]$. $c[v/x]$ is strongly normalizing since if $e = \tilde{\mu}x.c$ we know that $\forall v \in \mathbf{fst}(\llbracket T_1 \to T_2 \rrbracket), c[v/x] \in SN$.
3. $\langle \lambda x.v'' \| v' \cdot e' \rangle \longrightarrow \langle v' \| \tilde{\mu}x.\langle v'' \| e' \rangle \rangle$. If $\lambda x.v'' \in \mathbf{fst}(\llbracket T \rrbracket)$ then $\lambda x.v'' \in \mathbf{fst}(\llparenthesis T \rrparenthesis)$. Similarly, if $v' \cdot e' \in \mathbf{snd}(\llbracket T \rrbracket)$ then $v' \cdot e' \in \mathbf{snd}(\llparenthesis T \rrparenthesis)$. By the assumption that $\llparenthesis T \rrparenthesis$ is a nucleus we know that $\langle \lambda x.v'' \| v' \cdot e' \rangle$ is strongly normalizing and so is $\langle v' \| \tilde{\mu}x.\langle v'' \| e' \rangle \rangle$.

□

Lemma 8. *If $(\!|T|\!)$ is a nucleus then $[\![T]\!]$ is reducibility candidate.*

Proof.
- **CR 1**: Orthogonality holds by Lemma 7.
- **CR 2**: Forward closure holds by Lemma 6.
- **CR 3**: Inclusion of (co-)variables holds by definition of $F_{(\!|T|\!)}$.
- **CR 4**: Saturation holds by the fact that $[\![T]\!]$ is a fixed point of $F_{(\!|T|\!)}$.

□

Lemma 9. *For all types T in $\mu\tilde{\mu}^{\rightarrow}$, $[\![T]\!]$ is a reducibility candidate.*

Proof. By structural induction on T.

- In the base case T is a type variable. By Lemma 8 it is enough to show that $(\!|T|\!) = (\emptyset, \emptyset)$ is a nucleus.
  1. $\emptyset$ and $\emptyset$ are internally strongly normalizing (vacuously).
  2. $\emptyset$ and $\emptyset$ are forward closed (vacuously).
  3. There are no terms of the form $\mu\alpha.c$ in $\emptyset$ and no co-terms of the form $\tilde{\mu}x.c$ in $\emptyset$ by the definition of the empty set.
  4. $(\emptyset, \emptyset)$ is orthogonal, that is $\forall v \in \emptyset, \forall e \in \emptyset, \langle v \| e \rangle \in SN$ holds vacuously.
- In the inductive case we have T is of the form $T_1 \rightarrow T_2$. By Lemma 8 it is enough to show that $(\!|T|\!)$ is a nucleus under the inductive hypothesis that $[\![T_1]\!]$ and $[\![T_2]\!]$ are reducibility candidates.
  1. $\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\}$ and $\{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\}$ are internally strongly normalizing.
     - Let v be a term such that $\forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])$. Since $[\![T_1]\!]$ is a reducibility candidate it must satisfy **CR 3** and so $x \in \mathbf{fst}([\![T_1]\!])$, therefore $v = v[x/x] \in \mathbf{fst}([\![T_2]\!])$. Since $[\![T_2]\!]$ is a reducibility candidate, by Lemma 1 we know it is internally strongly normalizing, so v must be strongly normalizing and therefore $\lambda x.v$ is strongly normalizing. Thus $\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\}$ is internally strongly normalizing.
     - Let $v \in \mathbf{fst}([\![T_1]\!])$ and $e \in \mathbf{snd}([\![T_2]\!])$. Since $[\![T_1]\!]$ and $[\![T_2]\!]$ are reducibility candidates by Lemma 1 they are internally strongly normalizing and so there for v and e are strongly normalizing so the co-term $v \cdot e$ is strongly normalizing. Thus $\{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\}$ is an internally strongly normalizing set.

2. $\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\}$ and $\{v{\cdot}e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\}$ are forward closed.
   - Let $\lambda x.v$ such that $\forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])$. Given any v_1 such that $\lambda x.v \longrightarrow v_1$ we know $v_1 = \lambda x.v_1'$ where $v \longrightarrow v_1'$. Now, given any $v' \in \mathbf{fst}([\![T_1]\!])$ we know $v[v'/x] \in \mathbf{fst}([\![T_2]\!])$. Since $v \longrightarrow v_1'$ we also know that $v[v'/x] \longrightarrow^\star v_1'[v'/x]$ Because $\mathbf{fst}([\![T_2]\!])$ is a reducibility candidate it must be forward closed, so $v_1'[v'/x] \in \mathbf{fst}([\![T_2]\!])$. This shows that $\forall v' \in \mathbf{fst}([\![T_1]\!]), v_1'[v'/x] \in \mathbf{fst}([\![T_2]\!])$. Thus, $\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\}$ is forward closed.
   - Since $[\![T_1]\!]$ and $[\![T_2]\!]$ are reducibility candidates $\mathbf{fst}([\![T_1]\!])$ and $\mathbf{snd}([\![T_2]\!])$ are forward closed and so is $\{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\}$.
3. By definition there are no terms of the form $\mu\alpha.c$ in $\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\}$ and no co-terms of the form $\tilde{\mu}x.c$ in $\{v{\cdot}e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\}$.
4. $(\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\}, \{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\})$ is orthogonal, that is $\forall v \in A, \forall e \in \{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\}, \langle v \| e \rangle \in SN$.

   To see why this is the case, recall that a command is strongly normalizing if every command it reduces to is strongly normalizing. Since, $\{\lambda x.v | \forall v' \in \mathbf{fst}([\![T_1]\!]), v[v'/x] \in \mathbf{fst}([\![T_2]\!])\}$ and $\{v \cdot e | v \in \mathbf{fst}([\![T_1]\!]), e \in \mathbf{snd}([\![T_2]\!])\}$ are internally strongly normalizing and forward closed, we need only consider head reductions (Lemma 6). For head reductions, consider a v such that $\forall v_1 \in \mathbf{fst}([\![T_1]\!]), v[v_1/x] \in \mathbf{fst}([\![T_2]\!])$, a $v' \in \mathbf{fst}([\![T_1]\!])$ and an $e \in \mathbf{snd}([\![T_2]\!])$. We wish to show that the command $\langle \lambda x.v \| v' \cdot e \rangle$ is strongly normalizing. The only possible head reduction is $\langle \lambda x.v \| v' \cdot e \rangle \longrightarrow \langle v' \| \tilde{\mu}x.\langle v \| e \rangle \rangle$. So, it is enough to show that $\langle v' \| \tilde{\mu}x.\langle v \| e \rangle \rangle \in SN$.

   We know that $\forall v_1 \in \mathbf{fst}([\![T_1]\!]), v[v_1/x] \in \mathbf{fst}([\![T_2]\!])$ and that $e \in \mathbf{snd}([\![T_2]\!])$. Since $[\![T_2]\!]$ is a reducibility candidate it is orthogonal, so $\forall v_1 \in \mathbf{fst}([\![T_1]\!]), \langle v[v_1/x] \| e \rangle \in SN$ which can be rewritten as $\forall v_1 \in \mathbf{fst}([\![T_1]\!]), \langle v \| e \rangle [v_1/x] \in SN$. Since $[\![T_1]\!]$ is a reducibility candidate it is saturated, so if $\forall v_1 \in \mathbf{fst}([\![T_1]\!]), \langle v \| e \rangle [v_1/x] \in SN$ then $\tilde{\mu}x.\langle v \| e \rangle \in \mathbf{snd}([\![T_1]\!])$. We also know that $v' \in \mathbf{fst}([\![T_1]\!])$ and since $[\![T_1]\!]$ is a reducibility candidate it is orthogonal, so $\langle v' \| \tilde{\mu}x.\langle v \| e \rangle \rangle \in SN$.

□

Theorem 2. *Soundness for $\mu\tilde{\mu}^{\rightarrow}$:*

1. *if $\Gamma \vdash v : T \mid \Delta$ is derivable then $\Gamma \vDash v : T \mid \Delta$;*
2. *if $\Gamma \mid e : T \vdash \Delta$ is derivable then $\Gamma \mid e : T \vDash \Delta$;*
3. *if $c : (\Gamma \vdash \Delta)$ is derivable then $c : (\Gamma \vDash \Delta)$.*

Proof. By mutually induction on the derivation.

- The variable rule

$$\frac{}{\Gamma, x : T \vdash x : T \mid \Delta}$$

  is sound if it is always the case that $\Gamma, x : T \vDash x : T \mid \Delta$. Expanding the definition, this says that $\forall \psi \in [\![\Gamma, x : T]\!], \psi \in [\![\Delta]\!], \psi(x) \in \mathbf{fst}([\![T]\!])$ which holds since $\forall \psi \in [\![\Gamma, x : T]\!], \psi(x) \in \mathbf{snd}([\![T]\!])$ by definition.

- The co-variable rule is dual to the variable rule.

$$\frac{}{\Gamma \mid \alpha : T \vdash \alpha : T, \Delta}$$

  is sound if it is always the case that $\Gamma \mid \alpha : T \vDash \alpha : T, \Delta$ which holds since by definition $\forall \psi \in [\![\alpha : T, \Delta]\!], \psi(\alpha) \in [\![T]\!]$.

- The cut rule

$$\frac{\Gamma \vdash v : T \mid \Delta \qquad \Gamma \mid e : T \vdash \Delta}{\langle v \| e \rangle : (\Gamma \vdash \Delta)}$$

  is sound if it is always the case that $\langle v \| e \rangle : (\Gamma \vDash \Delta)$ under the inductive hypothesis that $\Gamma \vDash v : T \mid \Delta$ and $\Gamma \mid e : T \vdash \Delta$. Let ψ be a substitution such that $\psi \in [\![\Gamma]\!]$ and $\psi \in [\![\Delta]\!]$. We wish to show that $\psi(\langle v \| e \rangle) \in SN$. By inductive hypothesis, we know that $\psi(v) \in \mathbf{fst}([\![T]\!])$ and that $\psi(e) \in \mathbf{snd}([\![T]\!])$. Further, we know by Lemma 9 that T is a reducibility candidate and so therefore is orthogonal. Thus, it must be the case that $\langle \psi(v) \| \psi(e) \rangle \in SN$ and $\langle \psi(v) \| \psi(e) \rangle = \psi(\langle v \| e \rangle)$.

- The output abstraction rule

$$\frac{c : (\Gamma \vdash \alpha : T, \Delta)}{\Gamma \vdash \mu\alpha.c : T \mid \Delta}$$

  is sound if it is always the case that $\Gamma \vDash \mu\alpha.c : T \mid \Delta$ whenever $c : (\Gamma \vDash \alpha : T, \Delta)$. Let ψ be a substitution such that $\psi \in [\![\Gamma]\!]$ and $\psi \in [\![\Delta]\!]$. We wish to show that $\psi(\mu\alpha.c) \in \mathbf{fst}([\![T]\!])$. By Lemma 9 that $[\![T]\!]$ is a reducibility candidate. Further, $\psi(\mu\alpha.c) = \mu\alpha.\psi(c)$. It is thus enough to show that $\forall e \in \mathbf{snd}([\![T]\!]), \psi(c)[e/\alpha] \in SN$. Let $e \in \mathbf{snd}([\![T]\!])$. We can construct the substitution, $\psi' = \psi, \alpha \mapsto e$. By definition, $\psi' \in [\![\Gamma]\!]$ and $\psi' \in [\![\alpha : T, \Delta]\!]$. Thus by the inductive hypothesis, we that that $\psi'(c) \in SN$. Since $\psi'(c) = \psi(c)[\alpha/e]$, we are done.

- The input abstraction rule

$$\frac{c : (\Gamma, x : T \vdash \Delta)}{\Gamma \mid \tilde{\mu} x : T.c \vdash \Delta}$$

  is dual to output abstraction and is sound by a symmetric argument.

- The right rule for implication

$$\frac{\Gamma, x : T \vdash v : S \mid \Delta}{\Gamma \vdash \lambda x.v : T \to S \mid \Delta}$$

  is sound if $\Gamma \vDash \lambda x.v : T \to S \mid \Delta$ holds under the inductive hypothesis that $\Gamma, x : T \vDash v : S \mid \Delta$. To show this, let ψ be a substitution such that $\psi \in [\![\Gamma]\!]$ and $\psi \in [\![\Delta]\!]$. We wish to show that $\psi(\lambda x.v) \in \mathbf{fst}([\![T]\!])$. For any $v' \in \mathbf{fst}([\![T]\!])$ we can define the new substitution $\psi' = \psi, x \mapsto v'$. By definition, we know that $\psi' \in [\![\Gamma, x : T]\!]$ and $\psi' \in [\![\Delta]\!]$. Thus, by inductive hypothesis, $\psi'(v) = \psi(v)[v'/x] \in [\![S]\!]$. Abstracting over the v' we see that $\forall v' \in \mathbf{fst}([\![T]\!]), \psi(v)[v'/\prime] \in \mathbf{fst}([\![S]\!])$ so $\psi(\lambda x.v) = \lambda x.\psi(v) \in \mathbf{fst}([\![T \to S]\!])$.

- The left rule for implication

$$\frac{\Gamma \vdash v : T \mid \Delta \qquad \Gamma \mid e : S \vdash \Delta}{\Gamma \mid v \cdot e : T \to S \vdash \Delta}$$

  is sound if $\Gamma \mid v \cdot e : T \to S \vDash \Delta$ holds under the inductive hypothesis that $\Gamma \vDash v : T \mid \Delta$ and $\Gamma \mid e : S \vDash \Delta$. That is, for any ψ such that $\psi \in [\![\Gamma]\!]$ and $\psi \in [\![\Delta]\!]$ we have $\psi(v \cdot e) \in \mathbf{snd}([\![T \to S]\!])$. This holds by since by definition $\{v \cdot e | v \in \mathbf{fst}([\![T]\!]), e \in \mathbf{snd}([\![S]\!])\} \subseteq \mathbf{snd}([\![T \to S]\!])$, $\psi(v \cdot e) = \psi(v) \cdot \psi(e)$ and by inductive hypothesis $\psi(v) \in \mathbf{fst}([\![T]\!])$ and $\psi(e) \in \mathbf{snd}([\![S]\!])$.

□

Corollary 1. *Strong Normalization: If $c : \Gamma \vdash \Delta$ is provable in $\mu\tilde{\mu}^{\to}$ then c is strongly normalizing.*

Proof. By soundness, if $c : \Gamma \vdash \Delta$ is provable then $c : (\Gamma \vDash \Delta)$. Since $[\![-]\!]$ always yields a reducibility candidate (Lemma 9), the identity substitution is in $[\![\Gamma]\!]$ and $[\![\Delta]\!]$ so $id(c) = c \in SN$. □

5.1 Related Work in Normalization

Strong normalization for $\mu\tilde{\mu}^{\to,\forall,\exists}$ is closely related to the problem of cut elimination for classical second order logic (indeed, a cut elimination theorem can be derived from our proof of SN). Cut elimination for second order logic was posed as an open problem by

Takeuti [16] in the early nineteen fifties, and first shown approximately a decade later by Tait [15].

Reducibility candidates was originally discovered by Girard [7]. The exact details of what properties are necessary for a reducibility candidate vary, and, can seem rather ad-hoc. One hope of our variant for $\mu\tilde{\mu}^{\rightarrow,\forall,\exists}$ is that it makes these conditions obvious. In particular, while Girard's first two conditions on reducibility candidates are essentially identical to ours, his third condition is framed in terms of "neutral terms" which include a wide variety of terms in the lambda calculus. In our presentation however, Girard's "neutral terms" are reinterpreted as being two sorts of things: variables and terms of the form $\mu\alpha.c$.

Our variant of the reducibility candidates method constructs candidates by means of a fixed point construction similar to the Symmetric Candidates method of Barbanera and Berardi [1]. Lengrand and Miquel also used a variant of Symmetric Candidates to prove strong normalization for a calculus very similar to our $\mu\tilde{\mu}_S^{\overrightarrow{\ },\forall}$ (but extended with polymorphism at higher-kinds in the style of System F_ω). In particular, they showed the apparent necessity of the fixed point construction and proved that the common technique of using orthogonality alone won't work for non-confluent classical calculi [9]. The fixed point construction presented here was developed independently of the Symmetric Candidates method. As a result, the methods differ in non essential ways. For example, our justification for monotonicity in terms of a lattice of type-like-things consistent with the usual notion of sub-typing differs from the traditional justification in Symmetric Candidates which relies on the fact that the composition of two order reversing functions is monotone– this is though a difference only in presentation and intuition as the two arguments appear to be equivalent.

6 Conclusion

The term assignment for the polymorphic classical sequent calculus provides an alternative to traditional calculi based on natural deduction. The use of sequent calculus clarifies dualities, including the duality between existential and universal quantification. As future work, we are interested in applying the classical sequent calculus to the design of intermediate languages for compilers. In particular, we are exploring the possibility of using the sequent calculus as a new intermediate language in the Glasgow Haskell Compiler (GHC). Because GHC uses a typed intermediate language and supports polymorphism, the theory of quantification in the sequent calculus is essential to these efforts. In the context of compilers, strong normalization properties are interesting because they ensure that rewriting terminates. Of course, languages such as Haskell incorporate arbitrary recursion and are not strongly normalizing. It would be interesting to explore the possibility of extending the sequent calculus with unbounded recursion in such way that an easily recognized subset of the language remains strongly normalizing.

References

[1] Barbanera, F., and Berardi, S. A symmetric lambda calculus for "classical" program extraction. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software* (London, UK, UK, 1994), TACS '94, Springer-Verlag, pp. 495–515.

[2] Curien, P.-L., and Herbelin, H. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 233–243.

[3] Downen, P., and Ariola, Z. M. The duality of construction. In *ESOP* (2014), Z. Shao, Ed., vol. 8410 of *Lecture Notes in Computer Science*, Springer, pp. 249–269.

[4] Dreyer, D. Progress and preservation considered boring: A paean to parametricity. In *Programming Languages Mentoring Workshop* (2014).

[5] Felleisen, M., Friedman, D., and Kohlbecker, E. A syntactic theory of sequential control. *Theoretical Computer Science 52(3)* (1987), 205–237.

[6] Gentzen, G. Investigations into logical deduction. *American philosophical quarterly 1*, 4 (1964), 288–306.

[7] Girard, J.-Y., TAYLOR, P., and LAFONT, Y. *Proofs and types*, web reprint (2003) ed. Cambridge University Press, 1989.

[8] Lengrand, S., Dyckhoff, R., and McKinna, J. A sequent calculus for type theory. In *CSL 2006. LNCS* (2006), Springer.

[9] Lengrand, S., and Miquel, A. Classical fω, orthogonality and symmetric candidates. *Annals of Pure and Applied Logic 153*, 1 (2008), 3–20.

[10] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[11] Mitchell, J. C., and Plotkin, G. D. Abstract types have existential type. *ACM Trans. Program. Lang. Syst. 10*, 3 (July 1988), 470–502.

[12] Pierce, B. *Types and Programming Languages*. MIT Press, Cambridge, Massachesetts, 2002.

[13] Plato, J. v. Gentzen's proof of normalization for natural deduction. *Bulletin of Symbolic Logic 14*, 2 (06 2008), 240–257.

[14] Summers, A. J., and van Bakel, S. Approaches to polymorphism in classical sequent calculus. In *Proceedings of the 15th European Conference on Programming Languages and Systems* (Berlin, Heidelberg, 2006), ESOP'06, Springer-Verlag, pp. 84–99.

[15] Tait, W. W., et al. A non constructive proof of gentzen's hauptsatz for second order predicate logic. *Bulletin of the American Mathematical Society 72*, 6 (1966), 980–983.

[16] Takeuti, G. On a generalized logic calculus. *Japanese Journal of Mathematics 23* (1953), 39–96.

[17] Tarski, A. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics 5*, 2 (1955), 285–309.

[18] Wadler, P. The girard–reynolds isomorphism. *Inf. Comput. 186*, 2 (Nov. 2003), 260–284.